kProlog: an Algebraic Prolog for Kernel Programming

Francesco Orsini[1,2], Paolo Frasconi[2], and Luc De Raedt[1]

[1] Department of Computer Science, Katholieke Universiteit Leuven
{francesco.orsini, luc.deraedt}@cs.kuleuven.be
[2] Department of Information Engineering, Universitá degli Studi di Firenze
paolo.frasconi@unifi.it

Abstract. kProlog is a simple algebraic extension of Prolog with facts and rules annotated with semiring labels. We propose kProlog as a language for learning with kernels. kProlog allows to elegantly specify systems of algebraic expressions on databases. We propose some code examples of gradually increasing complexity, we give a declarative specification of some matrix operations and an algorithm to solve linear systems. Finally we show the encodings of state-of-the-art graph kernels such as Weisfeiler-Lehman graph kernels, propagation kernels and an instance of Graph Invariant Kernels (GIKs), a recent framework for graph kernels with continuous attributes. The number of feature extraction schemas, that we can compactly specify in kProlog, shows its potential for machine learning applications.

Keywords: graph kernels, Prolog, machine learning

1 Introduction

Statistical relational learning and probabilistic programming have contributed many declarative languages for supporting learning in relational representations. Prominent examples include Markov Logic [15], PRISM [16], Dyna [2] and ProbLog [1]. While these languages typically extend logical languages with probabilistic reasoning, there also exist extensions of a different nature: Dyna and aProbLog [9] are algebraic variations of probabilistic logical languages, while kLog [6] is a logical language for kernel-based learning.

Probabilistic languages such as PRISM and ProbLog label facts with probabilities, whereas Dyna and aProbLog use algebraic labels belonging to a semiring. Dyna has been used to encode many AI problems, including a simple distribution semantics, but does not support the disjoint-sum problem as ProbLog and aProbLog. While there has been a lot of research on integrating probabilistic and logic reasoning, kernel-based methods with logic have been much less investigated except for kLog and kFOIL [10]. kLog is a relational language for specifying kernel-based learning problems. It produces a graph representation of the learning problem in the spirit of knowledge-based model construction and then employs a graph kernel on the resulting representation. kLog was designed

to allow different graph kernels to be plugged in, but support to declaratively specify the exact form of the kernel is missing. kFOIL is a variation on the rule learner FOIL [14], that can learn kernels defined as the number of clauses that fire in both interpretations.

In the present paper, we investigate whether it is possible to use algebraic Prolog such as Dyna and aProbLog for kernel based learning. The underlying idea is that the labels will capture the kernel part, and the logic the structural part of the problem. Furthermore, unlike kLog and kFOIL, such a kernel based Prolog would allow to declaratively specify the kernel. More specifically, we propose kProlog, a simple algebraic extension of Prolog, where kProlog facts are labeled with semiring elements. kProlog introduces meta-functions that allow to use different semirings in the same program and overcomes the limited expressiveness of semiring sum and product operations.

kProlog can in principle handle the disjoint-sum problem as ProbLog and aProbLog, however in kernel design logical disjunctions and conjunctions are less common than algebraic sums and products, that are needed to specify matrix and tensor operations. We draw a parallel between kProlog and tensors showing how to encode matrix operations in a way that is reminiscent of tensor relational algebra [8]. Nevertheless kProlog supports recursion and so it is more expressive than tensor relational algebra.

We also show that kProlog can be used for specifying or programming kernels on structured data in a declarative way. We use polynomials as kProlog algebraic labels and show how they can be employed to specify label propagation and feature extraction schemas such as those used in recent graph kernels such as Weisfeiler-Lehman graph kernels [17], propagation kernels [12] and graph kernels with continuous attributes such as GIKs [13]. Polynomials were previously used in combination with logic programming for sensitivity analysis by Kimmig *et al.* (2011) and for data provenance by Green *et al.* (2007).

2 kProlog^S

We propose kProlog^S, an algebraic extension of Prolog in which facts and rules can be labeled with semiring elements.

Definition 1. *A kProlog^S program P is a 4-tuple (F, R, S, ℓ) where:*

- *F is a finite set of facts,*
- *R is a finite set of definite clauses (also called rules),*
- *S is a semiring with sum $\oplus$ and product $\otimes$ operations, whose neutral elements are 0_S and 1_S respectively,*[3]
- *$\ell : F \to S$ is a function that maps facts to semiring values.*

[3] A semiring is an algebraic structure $(\mathcal{S}, \oplus, \otimes, 0_S, 1_S)$ where $\mathcal{S}$ is a set equipped with sum $\oplus$ and product $\otimes$ operations. Sum $\oplus$ and product $\otimes$ are associative and have as neutral element 0_S and 1_S respectively. The sum $\oplus$ is commutative, multiplication distributes w.r.t addition and 0_S is the annihilating element of multiplication.

We use the syntactic convention α`::f` for algebraic facts where $f \in F$ is a fact and $\alpha = \ell(f)$ is the algebraic label.

Definition 2. *An algebraic interpretation $I_w = (I, w)$ of a ground kProlog^S program P with facts F and atoms A is a set of tuples $(a, w(a))$ where a is an atom in the Herbrand base A and $w(a)$ is an algebraic formula over the fact labels $\{\ell(f)|f \in F\}$. We use the symbol $\emptyset$ to denote the empty algebraic interpretation, i.e. $\{(true, 1_S)\} \cup \{(a, 0_S)|a \in A\}$.*

We use an adaptation of the notation used by Vlasselaer *et al.* (2015) in this definition and below.

Definition 3. *Let P be a ground algebraic logic program with algebraic facts F and atoms A. Let $I_w = (I, w)$ be an algebraic interpretation with pairs $(a, w(a))$. Then the $T_{(P,S)}$-operator is $T_{(P,S)}(I_w) = \{(a, w'(a))|a \in A\}$ where:*

$$w'(a) = \begin{cases} \ell(a) & \text{if } a \in F \\ \bigoplus\limits_{\substack{\{b_1,\ldots,b_n\} \subseteq I \\ a:-b_1,\ldots,b_n}} \bigotimes\limits_{i=1}^{n} w(b_i) & \text{if } a \in A \setminus F \end{cases} . \tag{1}$$

The least fixed point can be computed using a semi-naive evaluation. When the semiring is non-commutative the product $\otimes$ of the weights $w(b_i)$ must be computed in the same order that they appear in the rule. kProlog^S can represent matrices that in principle can have infinite size and can be indexed by using elements of the Herbrand universe of the program. We now show some elementary kProlog^S programs that specify matrix operations:

	algebra	kProlog^S	numerical example

matrix A	A	`1::a(0, 0).`
		`2::a(0, 1).`
		`3::a(1, 1).`
			$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right]$

matrix B	B	`2::b(0, 0).`
		`1::b(0, 1).`
		`5::b(1, 0).`
		`1::b(1, 1).`
			$\left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right]$

matrix transpose	A^t	`c(I, J) :- a(J, I).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right]^t = \left[\begin{smallmatrix} 1 & 0 \\ 2 & 3 \end{smallmatrix}\right]$
matrix sum	$A + B$	`c(I, J) :- a(I, J).` `c(I, J) :- b(I, J).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right] + \left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 3 & 3 \\ 5 & 4 \end{smallmatrix}\right]$
matrix product	AB	`c(I, J) :-` `  a(I, K), b(K, J).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right]\left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 12 & 3 \\ 15 & 3 \end{smallmatrix}\right]$
Hadamard product	$A \odot B$	`c(I, J) :-` `  a(I, J), b(I, J).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right] \odot \left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 2 & 2 \\ 0 & 3 \end{smallmatrix}\right]$
Kronecker product	$\text{kron}(A, B)$	`c(i(Ia, Ib), j(Ja, Jb)):-` `  a(Ia, Ja), b(Ib, Jb).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right] \otimes \left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 2 & 1 & 4 & 2 \\ 5 & 1 & 10 & 2 \\ 0 & 0 & 6 & 3 \\ 0 & 0 & 15 & 3 \end{smallmatrix}\right]$

The function symbols `i/2` and `j/2` were used to create the new indices that are needed by the Kronecker product. These definitions of matrix operations are reminiscent of tensor relational algebra [8]. Each of the above programs can be

evaluated by applying the $T_{(P,S)}(I_w)$ operator only once. For each program we have a different definition of the C matrix that is represented by the predicate `c/2`. As a consequence of Equation 1 all the algebraic labels of the `c/2` facts are polynomials in the algebraic labels of the `a/2` and `b/2` facts. We draw an analogy between the representation of a sparse tensor in coordinate format and the representation of an algebraic interpretation. A ground fact can be regarded as a tuple of indices/domain elements that uniquely identifies the cell of a tensor, the algebraic label of the fact represents the value stored in the cell. In kProlog$^{\mathrm{S}}$ for every atom a in the Herbrand base A the negation of a in an interpretation I_w can either be expressed with a sparse representation, by excluding it from the interpretation (i.e. $a \notin I_w$) or with a dense representation, including it in the interpretation with algebraic label 0_S (i.e. $a \in I_w$ and $w(a) = 0_S$).

Definition 4. *An algebraic interpretation $I_w = (I, w)$ is the fixed point of the $T_{(P,S)}(I_w)$-operator if and only if for all $a \in A$, $w(a) \equiv w'(a)$, where $w(a)$ and $w'(a)$ are algebraic formulae for a in I_w and $T_{(P,S)}(I_w)$ respectively.*

We denote with $T^i_{(P,S)}$ the function composition of $T_{(P,S)}$ with itself i times.

Corollary 1 (application of Kleene's theorem). *If S is an ω-continuous semiring*[4] *the algebraic system of fixed-point equations $I_w = T_{(P,S)}(I_w)$ admits a unique least solution $T^\infty_{(P,S)}(\emptyset)$ with respect to the partial order $\sqsubseteq$ and $T^\infty_{(P,S)}(\emptyset)$ is the supremum of the sequence $T^1_{(P,S)}(\emptyset), T^2_{(P,S)}(\emptyset), \ldots, T^i_{(P,S)}(\emptyset)$. So $T^\infty_{(P,S)}(\emptyset)$ can be approximated by computing successive elements of the sequence. If the semiring satisfies the* ascending chain property *(see [5]) then $T^\infty_{(P,S)}(\emptyset) = T^i_{(P,S)}(\emptyset)$ for some $i \geq 0$ and $T^\infty_{(P,S)}(\emptyset)$ can be computed exactly [5].*

We used $\emptyset$ to denote an empty algebraic interpretation. Examples of ω-continuous semirings are the boolean semiring ($\{\mathrm{T}, \mathrm{F}\}$, $\vee$, $\wedge$, F, T), the tropical semiring ($\mathbb{N} \cup \{\infty\}$, min, $+$, ∞, 0) and the fuzzy semiring ($[0, 1]$, max, min, 0, 1) [7]. Let us consider the following kProlog^S program:

```
1::edge(a, b).
3::edge(b, c).
7::edge(a, c).

path(X, Y):-
  edge(X, Y).
path(X, Y):-
  edge(X, Z), path(Z, Y).
```

If S is the tropical semiring, we obtain a specification of the Floyd-Warshall algorithm for all-pair shortest paths on graphs. Assuming that S is the boolean semiring and all the algebraic labels that are different from 0_S correspond to `true`$\in S$, we obtain the Warshall algorithm for the transitive closure of a binary relation. Lehmann (1977) explains how the Floyd-Warshall algorithm can be employed to invert square matrices. The inverse A^{-1} of a square matrix A can be computed as the result of the transitive closure of $I - A$ where I is the identity matrix. The last example requires the capability to compute additive inverses which are not guaranteed to exist for semirings.

[4] A ω-continuous semiring is a naturally ordered semiring extended with an infinite summation-operator $\sum$. The natural order relation $\sqsubseteq$ on a semiring S is defined by

3 kProlog

kProlog generalizes kProlog^S allowing multiple semirings and meta-functions. The coexistence of multiple semirings in the same program requires the declaration of the semiring of each algebraic predicate with the directive:

```
:- declare(<predicate>/<arity>, <semiring>).
```

We introduce meta-functions and meta-clauses to overcome the limits imposed by the semiring sum and product operations.

Definition 5 (meta-function). *A meta-function* `m`: $S_1 \times \ldots \times S_m \to S'$ *is a function that maps m semiring values $x_i \in S_i,\ i = 1, \ldots, k$ to a value of type S', where $S_1, \ldots, S_k$ and S' can be distinct semirings. Let* `a_1,...,a_k` *be algebraic atoms, the syntax* `@m[a_1,...,a_k]` *expresses that the meta-function* `@m` *is applied to the semiring values $w($`a_1`$), ..., w($`a_k`$)$ of the atoms* `a_1,...,a_k`.

Definition 6 (meta-clause). *In the kProlog language a meta-clause* `h :- b_1,...,b_n` *is a universally quantified expression where* `h` *is an atom and* `b_1,...,b_n` *can be either body atoms or meta-functions applied to other algebraic atoms. For a given meta-clause, if the head is labeled with the semiring S, also the labels of the body atoms and the return types of the meta-functions must be on the semiring S.*

Definition 7 (kProlog program). *A kProlog program P is a union of kProlog^{S_i} programs and meta-clauses.*

The introduction of meta-functions in kProlog allows us to deal with other algebraic structures such as rings that require the additive inverse `@minus/1` and fields that require the additive inverse and the multiplicative inverse `@inv/1`.

3.1 Recursive kProlog program with meta-functions

kProlog allows both additive and destructive updates, the update type is specified by the directive:

```
:- declare(<predicate>/<arity>, <semiring>, <update-type>).
```

where `update-type` can be either `additive` or `destructive`. [5]

We propose a simple example of an algebraic program that uses meta-functions to compute the limit $\lim_{n\to+\infty} g^n(x_0)$ of an iterated function $g(x) = x(1-x)$, where $g^n(x_0)$ of the function-composition of g with itself n times starting from some initial number x_0.

Assuming that $x_0 = 0.5$, in kProlog we would write:

$a \sqsubseteq b \Leftrightarrow \exists d \in S : a + d = b$. The semiring S is naturally ordered if $\sqsubseteq$ is a partial order on S; see [4, 3] for details.

[5] The directive `declare/3` must be used instead of `declare/2` whenever the groundings of the declared predicate appear in the cycles of the ground program. In case the

```
:- declare(x, real, destructive).
:- declare(x0, real).
0.5::x0.
x :- x0.
x :- @g[x].
```

The above program has the following behaviour: the weight $w(x)$ of `x` is initialised to $w(x_0) = 0.5$ and then updated at each step according to the rule $w'(x) = g(w(x))$. The directive `:- declare(x, real, destructive).` causes the result of the immediate-consequence operator to be used as a destructive assignment of the weights instead of an additive update.

We could have also considered an additive update rule such as $w'(x) = w(x) + g(w(x))$, but this would not lead us to the expected result for an iterated function system.

While iterated function systems require an update with destructive assignment, other programs such as the transitive closure of a binary relation (see above) or the compilation of ProbLog programs with SDDs require additive updates.[6]

3.2 The Jacobi method

We already showed that kProlog can express linear algebra operations, we now combine recursion and meta-functions in an algebraic program that specifies the Jacobi method. The Jacobi method is an iterative algorithm used for solving diagonally dominant systems of linear equations $A\mathbf{x} = \mathbf{b}$.

We consider the field of real numbers $\mathbb{R}$ (i.e. kProlog$^{\mathbb{R}}$) as semiring together with the meta-functions `@minus` and `@inv` that provide the inverse element of sum and product respectively.

The A matrix must be split according to the Jacobi method:

$D = \text{diag}(A)$ `d(I, I) :- a(I, I).`

$R = A - D$ `r(I, J) :- a(I, J), I \= J.`

The solution $\mathbf{x}^*$ of $A\mathbf{x} = \mathbf{b}$ is computed iteratively by finding the fixed point of $\mathbf{x} = D^{-1}(\mathbf{b} - R\mathbf{x})$. We call E the inverse of D. Since D is diagonal also E is a diagonal matrix:

$e_{ii} =$`invert`$(d_{ii}) = \frac{1}{d_{ii}}$ `e(I, I) :- @invert[d(I, I)].`

and the iterative step can be rewritten as $\mathbf{x} = E(\mathbf{b} - R\mathbf{x})$.

Making the summations explicit we can write:

$$x_i = e_{ik}\left(b_k - \sum_l r_{kl}x_l\right) \tag{2}$$

directive `declare/3` is not specified this can be detected by the system at evaluation time.

[6] The compilation of ProbLog programs [18] can be expressed in kProlog, provided

then we can extrapolate the term $\sum_l r_{kl} x_l$ turning it into the aux_k definition:

$$x_i = e_{ik}\,(b_k - aux_k)$$
$$aux_k = \sum_l r_{kl} x_l$$

```
:- declare(x/1, real, destructive).
:- declare(aux/1, real, destructive).
x(I) :-
  e(I, K), @subtraction[b(K), aux(K)].

aux(I) :-
  r(K, L),  x(L).
```

where `@subtraction/2` represents the subtraction between real numbers, `x/1` and `aux/1` are mutually recursive predicates. Because `x/1` needs to be initialized (perhaps at random) we also need the clause:

$$x_i = init_i \qquad \texttt{x(I) :- init(I).}$$

where `init/1` is a unary predicate. This example also shows that kProlog is more expressive than tensor relational algebra because it supports recursion.

3.3 kProlog T_P-operator with meta-functions

The algebraic T_P-operator of kProlog is defined on the meta-transformed program.

Definition 8 (meta-transformed program). *A meta-transformed kProlog program is a kProlog program in which all the meta-functions are expanded to algebraic atoms. For each rule* `h :- b_1,...,@m[a_1,...,a_k],...,b_n` *in the program P each meta-function* `@m[a_1,...,a_k]` *is replaced by a body atom* `b'` *and a meta-clause* `b':-@m[a_1,...,a_k]` *is added to the program P.*

Definition 9 (algebraic T_P-operator with meta-functions). *Let P be meta-transformed kProlog program with facts F and atoms A. Let $I_w = (I, w)$ be an algebraic interpretation with pairs $(a, w(a))$. Then the T_P-operator is $T_P(I_w) = \{(a, w'(a)) | a \in A\}$ where:*

$$w'(a) = \begin{cases} \ell(a) & \text{if } a \in F \\ \bigoplus\limits_{\substack{\{b_1,\ldots,b_n\}\subseteq I \\ a:-b_1,\ldots,b_n}} \bigotimes\limits_{i=1}^{n} w(b_i) \oplus \bigoplus\limits_{\substack{\{b_1,\ldots,b_k\}\subseteq I \\ a:-@m[b_1,\ldots,b_k]}} m(w(b_1),\ldots,w(b_k)) & \text{if } a \in A \setminus F\,. \end{cases} \tag{3}$$

The introduction of meta-functions makes the result of the evaluation of a kProlog program dependent on the order in which rules and meta-clauses are evaluated. For this reason we explain the order adopted by the kProlog language. A kProlog program P is grounded to a program ground(P) and then partitioned into a sequence of strata $P_1, \ldots, P_n$.

An atom in a non-recursive stratum P_i can only depend on the atoms from the previous strata $\bigcup_{j<i} P_j$, while an atom in a recursive stratum can depend on the atoms in $\bigcup_{j\leq i} P_j$. [7] Each partition P_i must be maximal and strongly connected (i.e. each atom in P_i depends on every other atom in P_i). The program evaluation starts by initializing the weight $w(a)$ of each ground atom a in ground(P) with 0_S where S is the semiring of the atom. Then the strata are visited in order and the weights are updated as follows: if the stratum P_i is non-recursive we apply the algebraic T_P-operator only once per atom, while if P_i is recursive we apply the algebraic T_P-operator only once for the non-recursive rules and meta-clauses and repeatedly until convergence for the recursive rules and meta-clauses.

When updating the weight $w(a)$ of a recursive atom a at each iteration we initialize a weight $\Delta w(a) = 0_s$. We accumulate on $\Delta w(a)$ the result of the application of the T_P-operator on all the recursive rules with head a. Then the new weight for a is computed as $w(a) = w(a) + \Delta w(a)$ or $w(a) = \Delta w(a)$ for additive and destructive update respectively.

If P_i is a cyclic stratum then the convergence of the algebraic T_P-operator must be guaranteed by the user that specifies the program. Nevertheless if the P_i is a cyclic stratum in which only rules are cyclic all the atoms in P_i are on the same semiring[8] S and so P_i has the same convergence properties of a kProlog$^{\mathrm{S}}$ program (see Corollary 1 on page 4). Whenever we apply the algebraic T_P-operator we use the Jacobi evaluation, so that the program is not affected by the order in which rules and meta-clauses are evaluated. This program evaluation procedure is an adaptation the work of Whaley *et al.* (2005) on Datalog and binary decision diagrams.

4 kProlog$^{S[\mathbf{x}]}$

kProlog$^{S[\mathbf{x}]}$ labels facts and rule heads with polynomials over the semiring S. kProlog$^{S[\mathbf{x}]}$ is a particular case of kProlog^{S} because polynomials over semirings are semirings in which addition and multiplication are defined as usual.

Definition 10 (Multivariate polynomials over commutative semirings). *A multivariate polynomial $\mathcal{P} \in S[\mathbf{x}]$ can be expressed as:*

$$\mathcal{P}(\mathbf{x}) = \bigoplus_{i=1}^{n} c_i \mathbf{x}^{\mathbf{e}_i} = \bigoplus_{i=1}^{n} c_i \otimes \bigotimes_{t \in T_i} x_t^{e_{it}} \tag{4}$$

where $c_i \in S$ are the coefficients of the i^{th} monomial and $\mathbf{x}$, $\mathbf{e}$ are vectors of variables and exponents respectively. The vector $\mathbf{x}$ is indexed by ground terms $t \in T$.

that the SDD semiring is used. The update of the algebraic weights must be additive, each update adds new proves for the ground atoms until convergence.

[7] We say that an atom **a** *directly depends* on an atom **b** if **a** is the head of a rule or a meta-clause and **b** is a body literal or an argument of a meta-function in the meta clause. We say that an atom **a** *depends* on an atom **b** either if **a** directly depends on **b** or there is an atom **c** such that **a** directly depends on **c** and **c** depends on **b**.

[8] atoms of distinct semirings cannot be mutually dependent without using meta-

4.1 Polynomials for feature extraction

We shall use polynomials to represent kernel features such as the ones computed by the Weisfeiler-Lehman and propagation kernels. We define an inner-product between multivariate polynomials of $\mathbb{R}[\mathbf{x}]$, with a finite number of monomials as:

$$\langle \mathcal{P}(\mathbf{x}), \mathcal{Q}(\mathbf{x}) \rangle = \sum_{(p,\mathbf{e}) \in \mathcal{P}} \sum_{(q,\mathbf{e}) \in \mathcal{Q}} pq. \tag{5}$$

For each monomial (uniquely identified by the vector of exponents $\mathbf{e}$) that appears in both the polynomials $\mathcal{P}$ and $\mathcal{Q}$, Equation 5 computes the product between their coefficients p and q respectively. These products are then summed together to obtain the value of the inner-product.

For example we can consider the multivariate polynomials on integer coefficients:

$$\begin{aligned} \mathcal{P}(x_1, x_2, x_3) &= 2x_1 + 3x_1x_2 + x_2x_3^2 \\ \mathcal{Q}(x_1, x_2, x_3) &= 4x_1 + 3x_1x_3 + 3x_2x_3^2 \end{aligned} \tag{6}$$

which can be expressed as two sets of coefficient-exponent pairs $\mathcal{P} = \{(2, [1,0,0]), (3, [1,1,0]), (1, [0,1,2])\}$ and $\mathcal{Q} = \{(4, [1,0,0]), (3, [1,0,1]), (3, [0,1,2])\}$ respectively. The two polynomials have in common the vectors of exponents $[1,0,0]$ and $[0,1,2]$, each contributes to the inner product by $2 \times 4 = 8$ and $1 \times 3 = 3$ respectively. The value of the inner product between $\mathcal{P}(x_1, x_2, x_3)$ and $\mathcal{Q}(x_1, x_2, x_3)$ is the sum of such contributes $8 + 3 = 11$.

In kProlog the inner-product between two algebraic atoms $\mathcal{P}(\mathbf{x})$`::a` and $\mathcal{Q}(\mathbf{x})$`::b` can be computed using the meta-function `@dot/2`. Another meta-function, that is useful for kernel design, is `@rbf/3`. The meta-function `@rbf/3` takes as input an atom labeled with a non-negative real value γ and two atoms labeled with the polynomials $\mathcal{P}$ and $\mathcal{Q}$ and computes the rbf kernel $exp\{-\gamma \|\mathcal{P} - \mathcal{Q}\|^2\}$.[9]

4.2 The @id meta-function

The `@id/1` meta-function `@id`: $S \to S$ is injective and transforms a polynomial $\mathcal{P}(\mathbf{x})$ to a new term t and returns the polynomial `@id`$[\mathcal{P}(\mathbf{x})] = 1.0 \cdot x(t)$. This function can be used to compress a multivariate polynomial to a new polynomial in a single variable. We use the `@id` meta-function for polynomial compression as Shervashidze *et al.* (2011) use the function f to compress multisets of labels.

Indeed we can represent a multiset μ of labels (we use Prolog ground terms to represent labels) as a polynomial:

$$\mathcal{P}_\mu(\mathbf{x}) = \sum_{t \in \mu} \sharp t \cdot x(t) \tag{7}$$

[9] The squared distance in the rbf kernel can be expressed by using the dot product, i.e. $\|\mathcal{P} - \mathcal{Q}\|^2 = \langle \mathcal{P}, \mathcal{P} \rangle + \langle \mathcal{Q}, \mathcal{Q} \rangle - 2\langle \mathcal{P}, \mathcal{Q} \rangle$

where $\sharp$ counts the number of occurrences of the label (identified by the ground term t) in the multiset μ.

Weisfeiler-Lehman algorithm: A colored graph G is a triple (V, E, ℓ) where V is a set of vertices, $E \subseteq V \times V$ is the set of the edges and $\ell : V \to \Sigma$ is a function that maps vertices to a color alphabet Σ. For example we can specify vertex labels and edge connectivity of a graph `graph_a` in kProlog as follows:

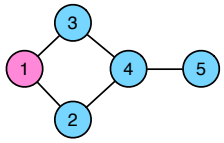
```
:- declare(vertex/2, polynomial(int)).
:- declare(edge_asymm/3, boolean).
:- declare(edge/3, polynomial(int)).

1 * x(pink)::vertex(graph_a, 1).
1 * x(blue)::vertex(graph_a, 2).
1 * x(blue)::vertex(graph_a, 3).
1 * x(blue)::vertex(graph_a, 4).
1 * x(blue)::vertex(graph_a, 5).

edge_asymm(graph_a, 1, 2).
edge_asymm(graph_a, 1, 3).
edge_asymm(graph_a, 2, 4).
edge_asymm(graph_a, 3, 4).
edge_asymm(graph_a, 4, 5).
```

```
1.0::edge(Graph, A, B):-
    edge_asymm(Graph, A, B).

1.0::edge(Graph, A, B):-
    edge_asymm(Graph, B, A).
```



where the boolean predicate `edge_asymm/3` is implicitly casted to integer and then to polynomial over integers when it appears in the definition of `edge/3`. The Weisfeiler-Lehman color of a vertex after h steps of the algorithm is defined as:

$$\mathcal{L}^h(v) = \begin{cases} \ell(v) & \text{if } h = 0 \\ f(\{\mathcal{L}^{h-1}(w) | w \in \mathcal{N}(v)\}) & \text{if } h > 0 \end{cases} \tag{8}$$

where $\mathcal{N}(v)$ is the set of the vertex neighbors of v an $\{\mathcal{L}^{h-1}(w) | w \in \mathcal{N}(v)\}$ is the multiset of their colors at step $h-1$. The Weisfeiler-Lehman algorithm can be specified in kProlog using the recursive definition of Equation 8:

```
:- declare(wl_color/3,
    polynomial(int)).
:- declare(wl_color_multiset/3,
    polynomial(int)).

wl_color_multiset(H, Graph, V):-
  edge(Graph, V, W),
  wl_color(H, Graph, W).
```

```
wl_color(0, Graph, V):-
  vertex(Graph, V).

wl_color(H, Graph, V):-
  H > 0,
  H1 is H - 1,
  @id[wl_color_multiset(H1, Graph, V)].
```

5 Graph Kernels

In this section we give the declarative specification of some recent graph kernels such as the Weisfeiler-Lehman graph kernel [17], propagation kernels [12] and graph invariant kernels [13]. These methods have been applied to different domains such as: natural language processing [13], computer vision [12] and bioinformatics [17, 12, 13].

5.1 Weisfeiler-Lehman graph kernel and Propagation kernels

The Weisfeiler-Lehman graph kernel is defined using a base kernel [17] that computes the inner-product between the histograms of Weisfeiler-Lehman colors of two graphs `Graph` and `GraphPrime`.

```
:- declare(phi/2, real).
phi(H, Graph):-
  wl_color(H, Graph, V).
```

```
:- declare(base_kernel/3, real).
base_kernel(H, Graph, GraphPrime):-
  @dot[phi(H, Graph),
       phi(H, GraphPrime)].
```

The Weisfeiler-Lehman graph kernel [17] with `H` iterations is the sum of base kernels computed for consecutive Weisfeiler-Lehman labeling steps $1, \ldots,$`H` on the graphs `Graph` and `GraphPrime`:

```
:- declare(kernel_wl/3, real).
kernel_wl(0, Graph, GraphPrime):-
  base_kernel(0, Graph, GraphPrime).

kernel_wl(H, Graph, GraphPrime):-
  H > 0, H1 is H - 1,
  kernel_wl(H1, Graph, GraphPrime).
```

```
kernel_wl(H, Graph, GraphPrime):-
  H > 0,
  base_kernel(H, Graph, GraphPrime).

.
```

Propagation kernels [12] are a generalization of the Weisfeiler-Lehman graph kernel, that can adopt different label propagation schemas. Neumann *et al.* (2012) implements propagation kernels using locality sensitive hashing. The kProlog specification is identical to the one the Weisfeiler-Lehman except that the `@id` meta-function is to be replaced with a meta-function that does locality sensitive hashing.

5.2 Graph invariant kernels

Graph Invariant Kernels (GIKs, pronounce "geeks") are a recent framework for graph kernels with continuous attributes [13]. GIKs compute a similarity measure between graphs G and G' matching them at vertex level according to the formula:

$$k(G, G') = \sum_{v \in V(G)} \sum_{v' \in V(G')} w(v, v') k_{\text{ATTR}}(v, v') \tag{9}$$

where $w(v, v')$ is the structural weight matrix and $k_{\text{ATTR}}(v, v')$ is a kernel on the continuous attributes of the graphs. We use `R`-neighborhood subgraphs, so the kProlog specification is parametrized by the variable `R`.

```
:- declare(gik_radius/3, real).
gik_radius(R, Graph, GraphPrime):-
  w_matrix(R, Graph, V, GraphPrime, VPrime),
  k_attr(Graph, V, GraphPrime, VPrime).
```

where `gik_radius/3`, `w_matrix/5` and `k_attr/4` are algebraic predicates on the real numbers semiring, which is represented with floats for implementation purposes. Assuming that we want to use the RBF with $\gamma = 0.5$ kernel on the vertex attributes we can write:

```
:- declare(rbf_gamma_const/0, real).
:- declare(k_attr/4, real).
0.5::rbf_gamma_const.
k_attr(Graph, V, GraphPrime, VPrime):-
  @rbf[rbf_gamma_const, attr(Graph, V), attr(GraphPrime, VPrime)].
```

where `attr/2` is an algebraic predicate that associates to the vertex `V` of a `Graph` a polynomial label. To associate to vertex `v_1` of `graph_a` the 4-dimensional feature $[1, 0, 0.5, 1.3]$ we would write:

```
:- declare(attr/2, polynomial(real)).
1.0 * x(1) + 0.5 * x(3) + 1.3 * x(4)::attr(graph_a, v_1).
```

while the meta-function `@rbf/3` takes as input an atom `rbf_gamma_const` labeled with the γ constant and the atoms relative to the vertex attributes.

The structural weight matrix $w(v, v')$ is defined as:

$$w(v, v') = \sum_{g \in \mathcal{R}^{-1}(G)} \sum_{g' \in \mathcal{R}^{-1}(G')} k_{\text{INV}}(v, v') \frac{\delta_m(g, g')}{|V_g||V_{g'}|} \mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}. \quad (10)$$

The weight $w(v, v')$ measures the structural similarity between vertices and is defined combining an $\mathcal{R}$-decomposition relation, a function $\delta_m(g, g')$ and a kernel on vertex invariants k_{INV} [13]. In our case the $\mathcal{R}$-decomposition generates `R`-neighborhood subgraphs (the same used in the experiments of Orsini *et al.* (2015)).

There are multiple ways to instantiate GIKs, we choose the version called LWL_V, because as shown with the experiments by Orsini *et al.* (2015), can achieve very good accuracies most of the times. LWL_V uses `R`-neighborhood subgraphs $\mathcal{R}$-decomposition relation, computes the kernel on vertex invariants $k_{\text{INV}}(v, v')$ at the pattern level (*local* GIK) and uses $\delta_m(g, g')$ to match subgraphs that have the same number of nodes.

In kProlog we would write:

```
:- declare(w_matrix/5, real).
w_matrix(R, Graph, V, GraphPrime, VPrime):-
  vertex_in_ball(Graph, R, BallRoot, V),
  vertex_in_ball(GraphPrime, R, BallRootPrime, VPrime),
  delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime),
  @inv[ball_size(R, Graph, BallRoot)],
  @inv[ball_size(R, GraphPrime, BallRootPrime)],
  k_inv(Graph, BallRoot, V, GraphPrime, BallRootPrime, VPrime).
```

where:

a) `vertex_in_ball(R, Graph, BallRoot, V)` is a boolean predicate which is true if `V` is a vertex of `Graph` inside a `R`-neighborhood subgraph rooted in `BallRoot`. `vertex_in_ball/4` encodes both the term $\mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}$ and the pattern generation of the decomposition relation $g \in \mathcal{R}^{-1}(G)$.

```
:- declare(vertex_in_ball/4, bool).
vertex_in_ball(0, Graph, Root, Root):-
  vertex(Graph, Root).

vertex_in_ball(R, Graph, Root, V):-
  R > 0, R1 is R - 1,
  vertex_in_ball(R, Graph, Root, V):-
    R > 0, R1 is R - 1,
    edge(Graph, Root, W),
    vertex_in_ball(R1, Graph, W, V).
```

b) `delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime)` matches subgraphs with the same number of vertices

```
:- declare(delta_match/5, real).
:- declare(v_id/3, polynomial(real)).
:- declare(ball_size/3, int).
```

```
delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime):-
  @eq[v_id(R, Graph, BallRoot), v_id(R, GraphPrime, BallRootPrime)].

v_id(R, Graph, BallRoot):- @id[ball_size(R, Graph, BallRoot)].

ball_size(R, Graph, BallRoot):- vertex_in_ball(R, Graph, BallRoot, V).
```

c) `@inv[ball_size(Radius, Graph, BallRoot)]` corresponds to the normalization term $1/|V_g|$. `@inv` is the meta-function that computes the multiplicative inverse and `ball_size(Radius, Graph, BallRoot)` is a the float predicate that counts the number of vertices in a `Radius`-neighborhood rooted in `BallRoot`.
d) `k_inv(R, Graph, BallRoot, V, GraphPrime, BallRootPrime, VPrime)` computes k_{INV} using `H_WL` iterations of the Weisfeiler-Lehman algorithm to obtain vertex features `phi_wl(R, H_WL, Graph, BallRoot, V)` from the `R`-neighborhood subgraphs.

```
:- declare(k_inv/7, real).
:- declare(phi_wl/5, polynomial(real)).
wl_iterations(3). % constant

k_inv(R, Graph, BallRoot, V, GraphPrime, BallRootPrime, VPrime):-
  wl_iterations(H_WL),
  @dot[phi_wl(R, H_WL, Graph, BallRoot, V),
    phi_wl(R, H_WL, GraphPrime, BallRootPrime, VPrime)].

phi_wl(R, 0, Graph, BallRoot, V):-
  wl_color(R, Graph, BallRoot, 0, V).
phi_wl(R, H, Graph, BallRoot, V):-
  H > 0, wl_color(R, Graph, BallRoot, H, V).

phi_wl(R, H, Graph, BallRoot, V):-
  H > 0, H1 is H-1,
  phi_wl(R, H1, Graph, BallRoot, V).
```

where `wl_color/5` is defined as `wl_color/3`, but has two additional arguments `R` and `BallRoot` that are needed to restrict the graph connectivity to the `R`-neighborhood subgraph rooted in vertex `BallRoot`.

6 Conclusions

We proposed kProlog, a simple algebraic extension of Prolog that can be used for kernel programming. Polynomials and meta-functions allow to elegantly specify in kProlog many recent kernels (e.g the Weisfeiler-Lehman Graph kernel, propagation kernels and GIKs). kProlog rules are used for kernel programming, but also to incorporate background knowledge and enrich the input data representation with user specified relations. kProlog is a language that provides a uniform representation for relational data, background knowledge and kernel design. In our future work we will exploit these three characteristics of kProlog to learn feature spaces with inductive logic programming.

Bibliography

[1] L De Raedt, A Kimmig, and H Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, 2007.

[2] J Eisner and N W Filardo. Dyna: Extending datalog for modern ai. In *Datalog Reloaded*. Springer, 2011.

[3] Javier Esparza and Michael Luttenberger. Solving fixed-point equations by derivation tree analysis. In *Algebra and Coalgebra in Computer Science*. Springer, 2011.

[4] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. An extension of newtons method to ω-continuous semirings. In *Developments in Language Theory*. Springer, 2007.

[5] J Esparza, M Luttenberger, and M Schlund. Fpsolve: A generic solver for fixpoint equations over semirings. In *Implementation and Application of Automata*. Springer, 2014.

[6] P Frasconi, F Costa, L De Raedt, and K De Grave. klog: A language for logical and relational learning with kernels. *Artificial Intelligence*, 2014.

[7] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems*. ACM, 2007.

[8] M Kim and K S Candan. Approximate tensor decomposition within a tensor-relational algebraic framework. In *Proceedings of the 20th ACM international Conference on Information and Knowledge Management*. ACM, 2011.

[9] A Kimmig, G Van den Broeck, and L De Raedt. An algebraic prolog for reasoning about possible worlds. In *25th AAAI Conference on Artificial Intelligence*, 2011.

[10] N Landwehr, A Passerini, L De Raedt, and P Frasconi. kfoil: Learning simple relational kernels. In *AAAI*, 2006.

[11] Daniel J Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 1977.

[12] M Neumann, N Patricia, R Garnett, and K Kersting. Efficient graph kernels by randomization. In *Machine Learning and Knowledge Discovery in Databases*. Springer, 2012.

[13] F Orsini, P Frasconi, and L De Raedt. Graph invariant kernels. In *Proceedings of the 24th IJCAI*, 2015.

[14] J. Ross Quinlan. Learning logical definitions from relations. *Machine learning*, 5(3):239–266, 1990.

[15] M Richardson and P Domingos. Markov logic networks. *Machine learning*, 2006.

[16] T Sato and Y Kameya. Prism: a language for symbolic-statistical modeling. In *IJCAI*, 1997.

[17] N Shervashidze, P Schweitzer, E J Van Leeuwen, K Mehlhorn, and K M Borgwardt. Weisfeiler-lehman graph kernels. *The Journal of Machine Learning Research*, 2011.

[18] J Vlasselaer, G Van den Broeck, A Kimmig, W Meert, and L De Raedt. Anytime inference in probabilistic logic programs with tp-compilation. In *Proceedings of the 24th IJCAI*, 2015.

[19] J Whaley, D Avots, M Carbin, and M S Lam. Using datalog with binary decision diagrams for program analysis. In *Programming Languages and Systems*. Springer, 2005.